Dátové štruktúry AVL, Splay, Hashovacie tabuľky

Patrik Boržík

Cvičiaci: Ing. Martin Komák, PhD.

26. marca 2023

Obsah

1 Hardvérové a softvérové vybavenie

V rámci použitých zdrojov na zrealizovanie projektu sme použili procesor Intel Core i5-8250U 1.60Hz, RAM pamäte v celkovom objeme 16GB s rýchlosťou zápisu 2400Hz – na danej pracovnej stanici sme používali OS Windows 10. Na samotné vyhotovenie kódu sme použili IDE od spoločnosti JetBrains, ide o IntelliJ IDEA. Celý kód je písaný v programovacom jazyku Java, kde využívame širšie spektrum funkcií rôznych vstavaných tried základných a často používaných knižníc ako Math alebo Random.

2 Dátové štruktúry a ich implementácia

Vytvorili sme program, ktorý implementuje 4 často používané dátové štruktúry – binárny vyhľadávací strom Splay, binárny vyhľadávací strom AVL a dve hlasovacie tabuľky, v ktorých používame rôzne spôsoby riešenia vzniknutých kolízií pri vytváraní hashovacích indexov v tabuľke. Implementovali sme základné operácie v daných štruktúrach a to vloženie prvku, zmazanie prvku a vyhľadanie chceného prvku na základe vopred určeného parametru. Princípy jednotlivých implementácií v jednotlivých štruktúr sú bližšie vysvetlené v nasledujúcich podkapitolách tejto kapitoly.

2.1 Binárny vyhľadávací strom - AVL

Ide o jednu zo základných dátových štruktúr, ktorej časová komplexnosť je v ideálnej situácií O(log n) a v najhoršej ide o lineárne závislosť, teda ide o O(n), kde samotná štruktúra stráca logickú aplikáciu, pretože ju považujeme za efektívnu ako spájaný zoznam. BST AVL vychádza s jednoduchej logiky, cieľom je udržať ľavú a pravú stranu tohto binárneho vyhľadávacieho stromu v rovnováhe vzhľadom na ich počet vetiev a prvkov na daných stranách. Tento spôsob udržiavania rovnováhy spočíva v takzvaných rotáciách, kde cieľavedome na základe istých možných situácií meníme poradie (v zmysle výšky a umiestnenia na danej vetve/výške) prvkov. Tieto rotácie delíme na 2 jednoduché a 2 zložené – ide o rotácia smerom doprava, smerom doľava, dvojitá rotácia doprava a dvojitá rotácia doľava alebo ich kombinácie.

2.1.1 Rotácie v AVL

Príklad pravej rotácie je v skrátenom príklade kódu nižšie. Rovnaký princíp premeny hodnôt sa aplikuje aj pri ľavej rotácií – v praktickej verzií kódu stačí premeniť všetky prvky s názvom „pravý“ za „ľavý“ a naopak. Viacnásobnou aplikáciou týchto dvoch jednoduchých rotácií vieme vytvoriť dvojité rotácie (v kóde sú následne volané dva-krát za sebou dané rotácie).

```
public static Node_avl rotaciaRight(Node_avl prvok){
        Node_avl a = prvok.lavy;
        Node_avl c = a.pravy;
        a.pravy=prvok;
        prvok.lavy=c;
        aktualizuj_vysku(prvok);
        aktualizuj_vysku(a);

        return a;
    }
```

2.1.2 Funkcia Prebalancovanie

Vstupnou premennou je prvok na prebalancovanie. Táto funkcia je kľúčová pri udržiavaní stromu, ktorý môžeme považovať za vyvážený. Jej úlohou je zistiť akú rotáciu je potrebné zvoliť pre daný prvok, ktorý zadáme ako vstupný parameter. Na vyhodnotenie situácie, daná funkcia prvotne aktualizuje výšku daného prvku, následne určí na základe rozdielu výšky pravej časti prvku a ľavej časti prvku o akú rotáciu pôjde. Následne podmienka určí či stačí vykonať jednu rotáciu alebo kombináciu dvoch rotácií v logickom slede na základe hodnosti výšok buď pravého „dieťaťa“ balancovaného prvku alebo ľavého „dieťaťa“ balancovaného prvku. V prípade, že index vyrovnávania je väčší ako 1 tak hovoríme, že je strom od prvku preťažený sprava a je potrebné vykonať ľavú rotáciu (alebo kombináciu pravej a ľavej kombinácie). Presne opačný postup operácií je v prípade, že spomínaný index je vyhodnotený ako menší než -1. Všeobecne považujeme strom za vyvážený ak nám daný index vychádza v celých číslach od -1 do 1. Následne funkcia „Prebalancovanie“ vráti už vyvážený prvok.

```
public static Node_avl Prebalancovanie(Node_avl prvok){
    aktualizuj_vysku(prvok);
    int balanc = Balancovanie(prvok);

    if(balanc > 1){
        if(vyska(prvok.pravy.pravy) > vyska(prvok.pravy.lavy)){
            prvok = rotaciaLeft(prvok);
        }else {
            prvok.pravy=rotaciaRight(prvok.pravy);
            prvok=rotaciaLeft(prvok);
            }
    }else if(balanc<-1){
        if(vyska(prvok.lavy.lavy) > vyska(prvok.lavy.pravy)){
                prvok=rotaciaRight(prvok);
        }else {
                prvok.lavy=rotaciaLeft(prvok.lavy);
                prvok=rotaciaRight(prvok);
            }
        }
        return prvok;
    }
```

2.1.3 Implementácia Insert - AVL

Vstupnými premennými je koreňový prvok stromu, hodnota nového prvku a reťazec znakov nového prvku. Princíp vloženia prvku do AVL stromu spočíva v lokalizácie správnej pozície prvku, ktorá vychádza z kľúčovej hodnoty daného prvku. Začíname porovnávať hodnotu, ktorú chceme vložiť s koreňom a následne po porovnaní určíme či sa daný prvok umiestni naľavo od koreňa alebo napravo (napravo väčšia hodnota než koreň a naľavo menšia hodnota než koreň). Samozrejme toto platí nie len pre koreň, ale aj pre nasledujúce prvky, ktoré sa už nachádzajú v strome. Rekurzívne prechádzame takto všetky prvky, ktoré určilo porovnávanie hodnôt, až do bodu, kde nasledujúce pravé či ľavé dieťa prvku má hodnotu null. Namiesto danej hodnoty null osadíme novo-vytvorený prvok. Následne spomínaná rekurzia zabezpečí, že každý z prvkov od novo-osadeného prechádza funkciou „Prebalancovanie“ a nakoniec vráti koreň, pod ktorým je koreňom už korektne vyrovnanému stromu.

```
public static Node_avl vloz(Node_avl prvok, int hodnota,String text)
    {
        if(prvok == null) {
            return new Node_avl(hodnota,text);
        }
        else if (prvok.data==hodnota) {
            //System.out.println("Duplikat");
            prvok.text="duplikat";
            return prvok;
        }
        else if (prvok.data<hodnota) {
            prvok.pravy= vloz(prvok.pravy,hodnota,text);
        }
        else {
            prvok.lavy=vloz(prvok.lavy,hodnota,text);
        }
        return Prebalancovanie(prvok);
    }
```

2.1.4 Implementácia Search - AVL

Vstupnými premennými sú koreňový prvok stromu a hodnota hľadaného prvku. Vyhľadávanie v rámci AVL binárnych stromov funguje relatívne jednoducho, funguje na hľadaní špecifickej hodnoty, ktorá vstupuje ako argument do danej funkcie. Prehľadávanie začína od koreňa stromu a následne na základe toho či hľadaná hodnota je väčšie alebo menšia než koreň, posúvame sa v hľadaní na ľavú alebo pravú vetvu od koreňa. Následne tento proces opakujeme pre prvok, na ktorý sme boli posunutý. Tento proces opakujeme dokým nenájdeme hľadanú hodnotu – ak sa v strome nachádza, tak touto logikou musí byť nájdená. Využívame while cyklus, avšak nato, aby sme si boli istý, že v prípade nenájdenia našej hodnoty sa nám nevytvoril nekonečný while cyklus, sme pridali aj „sledovanie“ výšky stromu od koreňa, ktorá sa dekrementuje s každou iteráciou cyklu – v prípade, že naše „sledovanie“ výšky klesne na hodnotu nula, vieme povedať,

že hodnota nebola nájdená a že sa nenachádza v danom strome. Následne nám vypíše hlášku ak ju potrebujeme či bol alebo nebol nájdený chcený prvok a v prípade, že bol nájdený funkcia nám vracia daný hľadaný prvok.

```
public static Node_avl najdi(Node_avl koren, int hladany){
        Node_avl pomocny=new Node_avl(0,null);
        int meranie_vysky= koren.vyska;
        pomocny=koren;

        while(pomocny.data!=hladany && meranie_vysky!=0) {
            if (hladany > pomocny.data) {
                pomocny=pomocny.pravy;
                meranie_vysky--;
                if(pomocny==null){break;}
            } else if (hladany < pomocny.data) {
                pomocny=pomocny.lavy;
                meranie_vysky--;
                if(pomocny==null){break;}
            }
        }
        if(pomocny!= null && pomocny.data!=hladany){
            System.out.println("Neexistuje daná hodnota v BST!");
            return koren;
        }else{
        /*System.out.println("Našlo sa!")*/ ;
        }
        return pomocny;
```

2.1.5 Implementácia Delete - AVL

Zmazanie prvku spočíva v princípoch, kde sa využíva veľká časť implementácie hľadania v AVL strome. Funkcia rovnakým princípom ako pri hľadaní nájde daný prvok alebo zhodnotí, že daný prvok sa nenachádza v binárnom strome. Potom ako nájde danú hodnotu, tak sa zhodnocuje situácia, ktorá určí ďalšie kroky metódy. Sú štyri prípady, ktoré môžu nastať – prvý prípad, kde prvok nemá ani jedného nasledovníka, potom dva podobné prípady, také kde chýba iba pravý alebo iba ľavý nasledovník (v tomto prípade sa ako náhrada za vymazaný prvok nahrádza ten daný jestvujúci nasledovník) a posledný prípad je ten, ktorý musíme vyriešiť, teda taký, kde prvok, ktorý chceme vymazať má oboch nasledovníkov. Riešime to tak, že nájdeme najväčší prvok v ľavom pod-strome, ktorý následne dosadíme ako náhradu za vymazaný. Nakoniec znova vyrovná prvky v rámci stromu a vracia prvok, ktorý slúži ako náhrada pôvodného.

```
public static Node_avl zmazanie(Node_avl prvok,int hodnota) {
    Node_avl max = null;
    if (prvok == null) {
        return prvok;
    } else if (hodnota > prvok.data) {
        prvok.pravy = zmazanie(prvok.pravy, hodnota);
    } else if (hodnota < prvok.data) {
        prvok.lavy = zmazanie(prvok.lavy, hodnota);
    } else if (prvok.data == hodnota) {
        if (prvok.lavy == null && prvok.pravy == null) {
            prvok = null;
        } else if (prvok.lavy == null && prvok.pravy != null) {
            prvok=prvok.pravy;
        } else if (prvok.lavy != null && prvok.pravy == null) {
            prvok = prvok.lavy;
        } else if (prvok.lavy != null && prvok.pravy != null) {
            max = najhlbsie(prvok.lavy);
            prvok.data = max.data;
            prvok.lavy=zmazanie(prvok.lavy,prvok.data);
        }
    }
    if(prvok!=null){
        Prebalancovanie(prvok);
    }
    return prvok;
```

```
}
```

2.2 Binárny vyhľadávací strom - Splay

Tento typ vyhľadávajúceho stromu je veľmi podobný predchádzajúcemu AVL stromu. Taktiež sa využívajú isté rotácie, avšak cieľom tohto stromu nie je udržiavať samého seba vyrovnaný, ale umiestňovať čo najbližšie ku koreňu tie prvky, ktoré bývajú čo najčastejšie používané – tento princíp je dodržiavaný tak, že každý prvok, ktorý je vložený alebo vyhľadaný je vždy uložený na miesto koreňa. Nepozeráme nato či strom je vyvážený.

2.2.1 Rotácie v Splay

V rámci rotácií v Splay BST používame taktiež ľavé a pravé rotácie, avšak zo zmenou použitia a aj pomenovania – taktiež sme pre implementovanie metód splayu nevyužívali rekurziu, čo nás doviedlo k riešeniu, kde sa odkazujeme na rodičovské prvky sledovaných prvkov. Variáciou rotácie doprava je tzv. rotácia Zig a variáciou rotácie doľava je tzv. rotácia Zag. Rovnako ako pri BST AVL taktiež môžeme tieto rotácie používať v istých situáciách samostatne, avšak potrebujeme aj využívať ich navzájom aplikované kombinácie pri cielene definovaných podmienkach. Príklad Zig rotácie je zobrazený pod touto kapitolou. Rovanko ako v príklade s AVL rotáciami, stačí nám zmeniť hodnoty pravy za lavy, to isté platí aj opačne - týmto získame variáciu rotácie Zag.

```
public static Node_splay Zig(Node_splay prvok){
    Node_splay a = prvok.lavy;
    Node_splay c = a.pravy;

    a.pravy=prvok;
    prvok.lavy=c;

    a.rodic = prvok.rodic;
    prvok.rodic=a;
    if(c!=null){
        c.rodic=prvok;
    }

    return a;
}
```

2.2.2 Funkcia Splaying

Funkcia Splaying je kľúčová v každej jednej z ďalších implementácií, pretože sa vykonáva s každou úpravou, vložením, hľadaním či zmazaním. Samotná implementácia tejto metódy spočíva v jednom širšom while cykle, ktorého podmienka stanovuje, že cyklus bude prebiehať dokým „splayovaný“ prvok nebude mať za svojho rodiča (predchodcu) hodnotu „null“ – jedine prvok, ktorý je koreň, nemá žiadneho rodiča, teda takto povieme cyklu, že daný „splayovaný“ prvok sa už úspešne koreňom stromu. Následne ak cyklus while zistí, že prvok ešte nie je koreňom daného stromu, prechádzame prvou podmienkou, ktorá rozdeľuje v akej situácií sa nachádza daný prvok a aký typ rotácie musíme aplikovať na tento daný prvok. Prvá podmienka zisťuje či sa daný „splayovaný“ prvok nachádza hneď pod koreňom a ak áno tak na základe toho, či sa nachádza sprava alebo zľava využijeme adekvátnu rotáciu. Ak tento stav nenastáva, tak sa presúvame na kontrolovanie ďalších 4 stavov. Prvé dva stavy, ktoré môžeme dosiahnuť sú také, že prvok a jeho rodič sú v jednej línií z jednej strany (ľavej alebo pravej) a potrebujeme vykonať rovnakú dvojnásobnú rotáciu Zig alebo Zag. Ďalšie dva prípady, ktoré môžu nastať sú takzvané Zig-Zag alebo Zag-Zig prípady, kde prvok sa nachádza z jednej strany voči svojmu rodičovskému prvku a daný rodičovský prvok sa nachádza presne z opačnej strany svojmu vlastnému rodičovskému prvku – o ktorý prípad ide filtrujeme pomocou vyraďovacích podmienok. Taktiež v rámci tejto implementácie sme sa stretávali s problémom, kde sa náš prvok, ktorý sme chceli „splayovať“ nie vždy samostatne pridal za svojho rodiča (teda až do bodu, dokým sa nestane koreňom), preto sme využili podmienkový systém, kde sa stane dieťaťom svojho rodiča podľa hodnosti, ktorú má vždy po komplikovanejších kombináciách rotácií. Nakoniec funkcia vracia prvok po vykonaní rotácií.

```
public static Node_splay splaying(Node_splay prvok){
    while(prvok.rodic!=null){
        if(prvok.rodic.rodic==null) {
            if (prvok.rodic.lavy == prvok ) {
                prvok = Zig(prvok.rodic);
            } else if (prvok.rodic.pravy == prvok) {
                prvok = Zag(prvok.rodic);}
        }else {
            if (prvok.rodic.rodic.lavy == prvok.rodic && prvok.rodic.lavy == prvok) {
```

```
                prvok = Zig(prvok.rodic.rodic);
                    prvok = Zig(prvok);
                 if(prvok.rodic!=null) {
                    if(prvok.data>prvok.rodic.data){
                            prvok.rodic.pravy=prvok;
                        }else {
                            prvok.rodic.lavy = prvok;}}
            } else if (prvok.rodic.rodic.pravy == prvok.rodic && prvok.rodic.pravy == prvok) {
                prvok = Zag(prvok.rodic.rodic);
                prvok = Zag(prvok);
                if(prvok.rodic!=null) {
                    if(prvok.data<prvok.rodic.data){
                        prvok.rodic.lavy=prvok;
                    }else {
                        prvok.rodic.pravy = prvok;}
                }
            } else if (prvok.rodic.rodic.pravy == prvok.rodic && prvok.rodic.lavy == prvok) {
                prvok = Zig(prvok.rodic);
                if(prvok.rodic!=null) {
                    if(prvok.data<prvok.rodic.data){
                        prvok.rodic.lavy=prvok;
                    }else {
                        prvok.rodic.pravy = prvok;
                    }
                }
                prvok = Zag(prvok.rodic);

                 if(prvok.rodic!=null) {
                   if(prvok.data<prvok.rodic.data){
                   prvok.rodic.lavy=prvok;
                }else {
                    prvok.rodic.pravy = prvok;}
            }
            } else if (prvok.rodic.rodic.lavy == prvok.rodic && prvok.rodic.pravy == prvok) {
                prvok = Zag(prvok.rodic);
                if(prvok.rodic!=null) {
                    if(prvok.data>prvok.rodic.data){
                        prvok.rodic.pravy=prvok;
                        }else {
                        prvok.rodic.lavy = prvok;}
                }
                prvok = Zig(prvok.rodic);

                if(prvok.rodic!=null) {
                    if(prvok.data>prvok.rodic.data){
                    rodic.pravy=prvok;
                    }else {
                        prvok.rodic.lavy = prvok;}
                }
            }
        }
    }
    return prvok;
}
```

2.2.3 Implementácia Insert - Splay

Vstupnými premennými je koreňový prvok stromu, hodnota nového prvku a reťazec znakov nového prvku. Ako aj bolo spomínané na začiatku tejto podkapitoly, BTS Splay a jeho implementácie sme robili bez rekurzie, teda aj táto implementácia vloženia prvkov vychádza na báze while cyklu, ktorého podmienka vychádza z toho, že prvok, ktorý vstupuje ako argument to danej funkcie nemá hodnotu null – každé vloženie je tento prvok koreňom celého stromu, teda iba ak ešte daný strom nemá koreň, while nevykoná ani jeden cyklus a automaticky vytvorí nový koreň, ktorý prechádza funkciou „splaying“ opísanú vyššie v tejto dokumentácií a následne vráti novo-vytvorený prvok na miesto

koreňa. V situácií, kde už využívame koreň s nie null hodnotou, funkcia funguje na princípe veľkostí hodnôt, ktoré chceme osadiť do nového vkladaného prvku rovnako ako v AVL strome. Teda ak daná hodnota je väčšia než koreň prvok bude posúvaný doprava od koreňa a ak bude menšia je posúvaná doľava – samozrejme, že rovnaký postup je aplikovaný po každom posune, až do bodu, že na mieste, kde chceme posunúť prvok je hodnota null namiesto prvku. V tomto bode vytvoríme prvok s vybranými hodnotami, uložíme predošlí prvok ako jeho rodiča a využívame ho ako argument to funkcie „splaying“, ktorá nám potom vráti prvok, ktorý môžeme následne dosadiť za nový koreň nášho stromu. Duplikáty rovnako ako pri AVL sú nájdené a nahrádzajú sa za pôvodný koreň, ktorý vstúpil do funkcie ako argument.

```
public static Node_splay vloz(Node_splay prvok,int hodnota, String text){
        Node_splay pomocny=prvok;
        Node_splay predosly=null;
        while(prvok!=null){
            if(hodnota>prvok.data){
                predosly=prvok;
                if(prvok.pravy==null) {
                    prvok.pravy = new Node_splay(hodnota, text);
                    prvok.pravy.rodic = predosly;
                    return splaying(prvok.pravy);
                }else {
                    prvok = prvok.pravy;
                }
            } else if (hodnota< prvok.data) {
                predosly=prvok;
                if(prvok.lavy==null){
                    prvok.lavy=new Node_splay(hodnota,text);
                    prvok.lavy.rodic=predosly;
                    return splaying(prvok.lavy);
                }else{
                    prvok=prvok.lavy;
                }
            }else
            {
                //System.out.println("Duplikat");
                return pomocny;
            }
        }
        prvok=new Node_splay(hodnota,text);
        prvok.rodic=predosly;
        return splaying(prvok);
    }
```

2.2.4 Implementácia Search - Splay

Tým, že ide stále o BST vieme použiť rovnaký algoritmus pre prehľadávanie aký sme aj využili pri AVL strome. Avšak aj keď princíp ostáva rovnaký, dopĺňame ho o to, že ak sa nájde daný prvok, bude slúžiť ako vstupný argument pre „splaying“ funkciu, ktorá ho dosadí za nový koreň stromu. Samotná funkcia má ako vstupné argumenty koreň stromu a hodnotu, ktorú chceme nájsť v strome. Následne začína kontrolou či náhodou koreň nemá hodnotu null alebo či hľadaná hodnota sa už náhodou nenachádza na mieste koreňa. Oba prípady sa riešia rovnakým spôsobom a to tak, že sa pôvodný vstupný koreň vráti. Ak ani jedna z podmienok neplatí, program začne prehľadávať strom na základe hľadanej hodnoty, teda ak je väčšia ako hodnota koreňového prvku posunie sa na pravý prvok a ak je menšia posunie sa na ľavý prvok. Tento cyklus dokým sa nenájde hľadaný prvok, ktorý je následné vložený do funkcie „splaying“ a dosadený za nový koreňový prvok alebo dokým sa cyklom nedostaneme do bodu, kde prehľadávaný prvok má hodnotu null, v tomto bode vieme, že prvok sa nenachádza v strome a vrátime prvok s hodnotou null.

```
public static Node_splay najdi(Node_splay koren, int hladany){
        Node_splay pomocny=null;
        if(koren==null ||koren.data==hladany){
            return koren;
        }

        while(koren.data!=hladany) {
            if (hladany > koren.data) {
                koren=koren.pravy;
```

```
                if(koren!=null) {
                    if (koren.data == hladany) {
                        //System.out.println("Našiel som! --> " + hladany);
                        return splaying(koren);
                    }
                }else{
                    System.out.println("Nenašlo sa!");
                    return  pomocny;}
            } else if (hladany < koren.data) {
                koren=koren.lavy;
                if(koren!=null) {
                    if (koren.data == hladany) {
                        //System.out.println("Našiel som! ");
                        return splaying(koren);
                    }
                }else {
                    System.out.println("Nenašlo sa!");
                    return pomocny;}
            }
        }

        return koren;
    }
```

2.2.5 Implementácia Delete - Splay

V rámci implementácie zmazania využívame funkciu vyhľadanie daného prvku, ktorý chceme zmazať – ako vstupné argumenty opätovne slúžia koreňový prvok a hodnota podľa, ktorej program nájde daný prvok. Následne si uložíme najväčší ľavý koreň prvku, ktorý bude slúžiť ako náhrada za vymazaný koreň, taktiež si uložíme aj pravý prvok od toho, ktorý chceme vymazať. Vymažeme starý koreň a nový koreň, ktorý je najväčší z ľavého podstromu pôvodného koreňa použijeme ako vstupný argument do funkcie „splaying“, následne pridáme pravý podstrom pôvodného koreňa ku novému. Táto funkcia nakoniec vráti nový koreň a umiestni za koreň celého stromu.

```
public static Node_splay zmazanie (Node_splay koren, int hladany){
        Node_splay na_vymazanie=najdi(koren,hladany);
        if(na_vymazanie==null){
            System.out.println("Prvok neexistuje!");
            return koren;
        }

        Node_splay novy_koren=na_vymazanie.lavy;
        Node_splay pravy_od_korena=na_vymazanie.pravy;

        na_vymazanie=null;

        novy_koren=splaying(najhlbsie(novy_koren));

        novy_koren.pravy=pravy_od_korena;
        if(pravy_od_korena!=null) {
            pravy_od_korena.rodic = novy_koren;
        }

        return novy_koren;
    }
```

2.3 Hashovacia tabuľka – Linear Probing

Ďalšou dátovou štruktúrou, ktorú sme implementovali boli hashovacie tabuľky, ktoré spočívajú v generovaní indexov na základe určitých matematických funkcií a vstupných dát. Indexy sú tvorené na základe hodnôt, čo znamená, že sa často môžeš stať situácia, kde dôjde k prekrývaniu. Presne tieto prípady musíme riešiť. Prvá zvolená metóda bola tzv. linear probing, kde ak dôjde ku kolízií indexov, posúvame index o 1 nahor, dokým nenájdeme voľné miesto. V rámci možnosti zväčšovania alebo zmenšovania tabuľky, každá tabuľka je inicializovaná na počet rovný polovici vkladaného počtu prvkov. .

2.3.1 Funkcia „hashujem adresovanie“

Táto funkcia je kľúčovým aspektom danej hash tabuľky, pretože jej účel je tvorenie indexov na základe reťazca znakov a veľkosti tabuľky, ktoré vstupujú do tejto funkcie ako parametre. Následne vygenerujeme index v podobe celého čísla. Na vytvorenie hashu používame vstavanú metódu „hashCode“, ďalej sa vytvorená hodnota násobí prvočíslom 31 a pomocou matematickej funkcie modulo veľkosťou tabuľky sa vygeneruje finálny index, ktorý nakoniec funkcia vráti. Existujú prípady, kde je vygenerovaná záporná hodnota, ktorá sa následne jednoducho dá do absolutnej hodnoty, aby bola použiteľná ako index v rámci tabuľky.

```
public static int hashujem_adresovanie(String text, int velkost_tabulky) {
        int hash = 0;

        hash = (text.hashCode() * 31) % velkost_tabulky;
        if (hash < 0) {
            hash = Math.abs(hash);
        }
        return hash;
    }
```

2.3.2 Implementácia Insert – Linear Probing

Vkladanie prvkov do tabuľky pomocou linear probingu je relatívne jednoduché, ako vstupné argumenty ma daná funkcia reťazec znakov, hodnotu (pre priradenie hodnoty prvku), pole prvkov, iteráciu vkladajúceho sa prvku a počet všetkých prvkov. Na začiatku sa vytvorí nový prvok, ktorý obsahuje reťazec znakov a celočíselnú hodnotu so vstupných argumentov. Následne sa vygeneruje index pomocou hashovacej funkcie. Ďalší riadky kódu overujú či je potrebné tabuľku zmenšovať alebo zväčšovať. Keď je tabuľka potrebnej veľkosti a máme aj určený index, skúsime dosadiť daný prvok na miesto v tabuľke, ak je daný index voľný, bez problémov je uložený na dané miesto, avšak pri kolízií (index nie je prázdny) sa posúva index o hodnotu 1 nahor, až dokým nenájde voľné miesto – v prípade, že sa dostane na „koniec“ tabuľky, nezväčšujeme tabuľku, ale vynulujeme index a skúšame či sa nenájde miesto na začiatku tabuľky. Po priradení miesta v tabuľke pre nový prvok, funkcia vracia novo osadenú tabuľku.

```
public static Node_hash[] vloz_adresovanie(String text, int hodnota, Node_hash[] array,
int iteracia, int pocet_prvkov){

        int index=0; int hash=0;
        Node_hash prvok=new Node_hash(hodnota,text);
        int velkost_tabulky=array.length;

        index=hashujem_adresovanie(prvok.text,velkost_tabulky);
        hash=index;

        if(index>array.length-1) {
            array = zvacsi_adresovanie(array, array.length, index,prvok);
            return array;
        }
        if(iteracia==pocet_prvkov/2){
            array=zmensi_adresovanie(array, array.length,index,prvok);
            return array;
        }
        while(array[index]!=null){
            if(array[index]!=null){
                index=index+1;
                if(index==hash){
                    array=zvacsi_adresovanie(array, array.length, array.length, prvok);
                    return  array;};
                if(index==array.length){index=0;}
            }
        }
        array[index]=prvok;
        return array;
    }
```

2.3.3 Zväčšovanie a zmenšovanie tabuľky

Zmenu veľkosti tabuľky začínajú dva aspekty počas priebehu vkladania nových prvkov do našej tabuľky. Prvý častejší prípad je, keď sa vygeneruje pomocou hashovania index, ktorý je väčší ako celková veľkosť tabuľky – v tomto prípade sa nám zväčšuje tabuľka na veľkosť rovnú danému indexu ešte aj s malou rezervou pre zmenšenie šancí na viacnásobné zväčšovanie za sebou – všetky prvky sa prehashujú do novo-vzniknutej tabuľky. Druhý prípad zmeny veľkosti tabuľky nastáva prirodzene v polovici priebehu vkladania prvkov do tabuľky a rovnako sa opakuje aj pri naplnení tabuľky 90% prvkami – kontroluje sa počet nulových (prázdnych) miest a ak prekročí viac než 30% percent pôvodnej tabuľky, zmenší sa o 25% a všetky prvky sa presádzajú a prehashujú do novo-vzniknutej tabuľky. Kvôli veľkosti kódov, dané kódy nie sú v dokumentácií priložené.

2.3.4 Implementácia Search - Linear Probing

Hľadanie v rámci hashovacích tabuliek funguje na podobnom princípe ako samotné vkladanie. Vstupné argumenty pre danú funkciu sú pole prvkov (tabuľka) a reťazec znakov. Vytvoríme index na základe vstupného reťazca písmen, ktorý rovnako ako pri vkladaní najprv skúsime vložiť hneď na index, ktorý bol vytvorený hashovaciou funkciou a ak nastane kolízia indexov lineárne sa posúvame až do bodu, dokým ho nenájdeme. Aby sme zistili to, že daný cyklus, ktorý prechádza tabuľkou neostal zacyklený v rámci podmienky kontrolujeme či nenájdeme index s hodnotou null alebo či sme už neprešli celú tabuľku a nevrátili sa späť na rovnaký index ako na začiatku. Ak sa vrátime späť na pôvodný index alebo nájdeme index s hodnotou null môžeme usúdiť, že daný prvok sa v tabuľke nenachádza. Tak isto to platí pre prípad, kde sa na danom indexe nájde nullové/vymazané miesto. Nakoniec funkcia vráti nájdený prvok alebo prvok, ktorý má hodnoty indikujúc, že prvok bol vymazaný/nikdy sa ani nenachádzal v danej tabuľke.

```
public static Node_hash hladaj_adresovanie(Node_hash[] array,String text){
        int hash=0;
        Node_hash hashulo=null;
        int index=hashujem_adresovanie(text,array.length);
        hash=index;

        hashulo=array[hash];
        if(hashulo==null){
            System.out.println("Prvok neexistuje");
            return null;}
        while(hashulo.text!=text){
            hash=hash+1;
            hashulo=array[hash];
            if(hash==array.length){hash=0;}
            if(hashulo==null){break;}

            if(hash==index || hashulo==null){
                System.out.println("Prvok neexistuje");
                return  null;}}
        return hashulo;
    }
```

2.3.5 Implementácia Zmazania – Linear Probing

Rovnako ako aj predošlé implementácie funkcií aj zmazávanie prvkov je založené na definovaní indexu, ktorý vychádza z hashovacej funkcie. Do tejto funkcie vstupujú dva argumenty a to pole prvkov (tabuľka) a reťazec znakov, ktorý chceme lokalizovať a vymazať. Po vygenerovaní indexu sa nám uloží do pomocnej premennej prvok, ktorý sa nachádza na danom indexe, ak je nulový vieme iste povedať, že určite sa ten prvok nenachádza v zozname. Ak tento prípad nenastane, tak prehľadávame tabuľku tak, že ideme od daného indexu vždy o +1 miesto, až dokým nenájdeme danú hodnotu alebo index s hodnotou null. Následne ak nájdeme prvok, ktorý chceme vymazať označíme ho za „vymazaný“, aby v prípade ďalšieho hľadanie/mazania nedošlo k predčasnému ukončeniu programu a tým pádom aj k chybe. Po zmene prvku na „vymazaný“ sa vráti prepísaná tabuľka alebo v prípade nájdenia null prvku vráti pôvodnú tabuľku.

```
public static Node_hash[] zmazanie_adresovanie (Node_hash[] array, String text){
        int index=hashujem_adresovanie(text, array.length);
        int hash=index;
        Node_hash hashulo=array[hash];

        if(hashulo==null){
            System.out.println("Prvok neexistuje");
            return array;
        }
```

```
        if(hashulo.text==text){
            array[hash]=new Node_hash(0,"VYMAZANY");
            return array;
        }else {
            while(hashulo.text!=text){
                hash=hash+1;
                hashulo=array[hash];
                if(hash>array.length){
                    hash=0;
                }
                if(hashulo==null || hash==index){
                    System.out.println("Prvok sa v tabuľke nenachádza!");
                    return array;
                }}
            array[hash]=new Node_hash(0,"VYMAZANY");
        }
        return array;
    }
```

2.4 Hashovacia tabuľka – metóda Chainingh

V rámci tohto typu hashovacej tabuľky princíp samotnej štruktúry je veľmi podobný ako pri iných hashovacích tabuľkách. Spočíva vo využívaní matematických funkcií a operácií na vygenerovanie indexu, kde sa daný prvok uloží/ je uložený do inicializovanej tabuľky - využívame rovnakú hashovaciu funkciu aj pri tejto metóde, z dôvodu efektívnej distribúcie indexov. Spôsob ako sa v tomto prípade riešia kolízie medzi ukladaním prvkov je taký, že sa vytvárajú kratšie spájané zoznamy medzi prvkami s rovnakým prvkom systému, kto prvý príde ten je vyššie v rámci vertikálnej reprezentácie spájaného zoznamu – teda prvý prvok je osadený ako „hlavička“ spájaného zoznamu a ďalšie prvky, ktoré sa chcú uložiť na daný index sú uložené ako deti hlavičky alebo následných prvkov v danom zozname.

Proces zmeny veľkosti tabuľky či zväčšenia, či zmenšenia ostáva rovnaký, aký sme použili aj pri metóde linear probingu.

2.4.1 Implementácia Insert – Chaining

V rámci implementácie funkcie vkladanie prvkov do danej tabuľky ako vstupné argumenty vstupujú reťazec znakov, hodnota pre nový prvok, pole prvkov (tabuľka), interácia vkladania a počet všetkých prvkov. Následne ako pri predošlom type hashovacej tabuľky sa generuje index na základe internej funkcie na hashovanie, do ktorej vkladáme všetky hodnoty ASCII tých znakov z reťazca, následné táto hodnota je vynásobená prvočíslom 31 a následne používame funkciu modulo s hodnotou veľkosti tabuľky. Tento index sa následne použije na vloženie novo-vytvoreného prvku do tabuľky, ak nastane situácia, že na danom indexe je už uložený prvok, osadíme ho za jeho „dieťa“ – toto aplikujem aj pri väčšom počte prvkov v rámci jedného index dokým sa nový prvok neosadí za hodnotu null. Po osadení daného prvku funkcia nám vracia novú tabuľku. Taktiež overujeme potrebu zväčšovania / zmenšovania tabuľky rovnakým spôsobom ako pri metóde linear probing.

```
    public static Node_hash[] vloz_chain(String text, int hodnota, Node_hash[] array,
    int iteracia, int pocet_prvkov){

        int index=0;
        Node_hash prvok_pomocny;
        Node_hash povodny;
        Node_hash prvok=new Node_hash(hodnota,text);
        int velkost_tabulky=array.length;

        index=hashujem_chain(prvok.text,velkost_tabulky);

        if(index>array.length-1) {
            array = zvacsi(array, array.length, index,prvok);
            return array;
        }
        if(iteracia==pocet_prvkov/2 || iteracia==(int)(pocet_prvkov*0.9)){
            array= zmensi(array, array.length, index, prvok);
            return array;
        }
        prvok_pomocny=array[index];
        povodny=prvok_pomocny;
```

```
        if(prvok_pomocny!=null){
            while (prvok_pomocny.kamos!=null){
                prvok_pomocny=prvok_pomocny.kamos;
            }
            prvok_pomocny.kamos=prvok;
            array[index]=povodny;
        }else {
            array[index]=prvok;
        }
        return array;
    }
```

2.4.2 Implementácia Search – Chaining

Táto implementácia sa zakladá na oveľa jednoduchšom princípe v porovnaní s predošlou metódou riešenia kolízií pri linear probingu. Funkcia na vyhľadávanie prvkov má ako vstupné argumenty pole prvkov (tabuľku) a reťazec znakov prvku, ktorý chceme nájsť. Následne na základe reťazca znakov sa vytvorí index, na ktorom by sa mal daný prvok v tabuľke nachádzať. Tento index sa skontroluje či nemá hodnotu null, v takom prípade vieme, že daný prvok sa v zozname nenachádza. Avšak sa nájde hodnota, ktorá nie je null na danom indexe, program prechádza všetky prvky v danom spájanom zozname až dokým nenájde daný hľadaný prvok, ktorý následne funkcia vráti. Druhá možnosť je, že prvok nenájde a v takomto prípade rovnako ako pri prvotnej hodnote null na vytvorenom indexe vieme, že daná prvok sa nenachádza v tabuľke, teda funkcia vráti hodnotu null (hľadaný prvok).

```
public static Node_hash hladaj_chain(Node_hash[] array,String text){
        int hash=0;
        Node_hash hashulo=null;
        hash=hashujem_chain(text, array.length);
        hashulo=array[hash];

        if(hashulo==null){
            return null;
        }
        while(hashulo.text!=text){
            hashulo=hashulo.kamos;
            if(hashulo==null){;
                return null;
            }
        }
        return hashulo;
    }
```

2.4.3 Implementácia Delete – Chaining

Rovnako môžeme konštatovať, že aj v prípade vymazávania prvkov pri využití metódy chaining je relatívne jednoduché v porovnaní s metódou linear probing. Vstupné argumenty pre funkciu na vymazávanie prvkov sú pole prvkov (tabuľka) a reťazec znakov, ktorý identifikuje prvok určený na vymazanie. Na základe vloženého reťazca sa vygeneruje index. Na danom indexe sa následné hľadá prvok na vymazanie, kde môžu vzniknúť tri prípady – prvok na vymazanie je hlavička spájaného zoznamu na danom indexe, druhá možnosť, že je jeden z prvkov okrem hlavičky v danom spájanom zozname alebo prvok na vymazanie sa vôbec nenachádza v tabuľke. Prvý prípad sa rieši tak, že dosadíme susedný najbližší prvok v spájanom zozname za prvok, ktorý chceme vymazať. Druhý prípad sa rieši tak, že spájaný zoznam sa začne prehľadávať a po nájdení chceného prvku sa nahradí opätovne vedľajším prvkom a dosadí sa celý spájaný zoznam opätovne na daný index v tabuľke. Posledný prípad nastane, keď sa prehľadá celý spájaný zoznam na danom indexe až do bodu, kde posledný ďalší prvok má hodnotu null – vtedy vieme, že prvok sa v tabuľke nenachádza a funkcia vracia pôvodnú tabuľku bez zmeny.

```
    public static Node_hash[] zmazanie_chain (Node_hash[] array, String text){
        int hash=hashujem_chain(text, array.length);
        Node_hash hashulo=array[hash];
        Node_hash povodny=hashulo;
         if(hashulo==null){
            return null;
        }
        if(hashulo.text==text){
```

```
            array[hash]=hashulo.kamos;
        }else {
            while (hashulo.kamos.text != text) {
                hashulo = hashulo.kamos;
                if(hashulo.kamos==null){;
                    return array;
                }
            }
            hashulo.kamos = hashulo.kamos.kamos;
            array[hash]=povodny;
        }
        return array;
    }
```

3 Zhodnotenie meraní

V rámci testovania sme sa zamerali na časovú zložitosť všetkých štruktúr – porovnávali sme, ako dlho trvá vložiť nový prvok pri hodnotách 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000 vopred vložených prvkov. Taktiež po vložení týchto hodnôt sme pozorovali ako dlho bude trvať nájdenie prvku, ktorý bol vložený v strede daného datasetu. Ďalej sme sa venovali ako dlho bude trvať mazanie stredného prvku pri vložení mocnín čísla 10 do hodnoty 10000000. Rovnako tak, sme pozorovali koľko pamäte sa využije pri napĺňaní daných dátových štruktúr pri rôznych počtoch vkladaných prvkov.

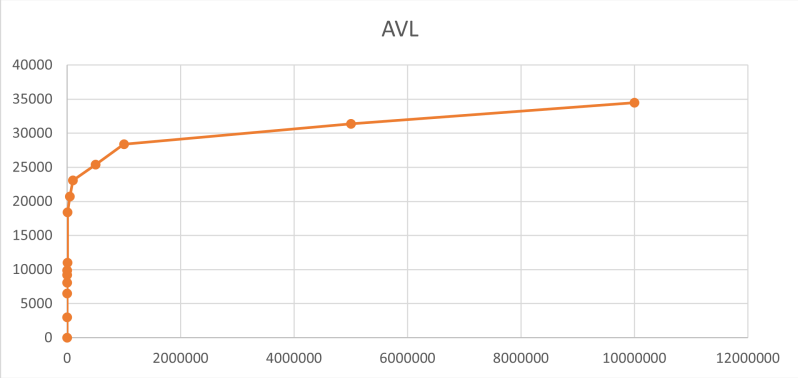
3.1 Testovacie súbory

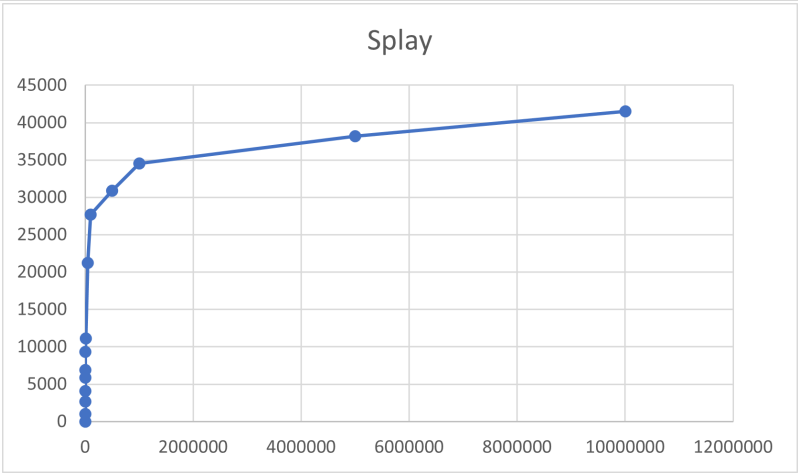
Na otestovanie funkčnosti kódu sme vytvorili a použili testovací súbor s názvom „Test2.java“, kde overujeme počty pribudnutých prvkov, či duplikátov. Druhý súbor s názvom „Test1.java“ je modifikovateľný súbor, kde sme vytvárali rôzne možnosti kombinácií implementácií, ktorým sme následne merali čas vykonania procesu. Údaje v danej kapitole boli spracované z spomínaného súboru „Test1.java“. Druhý testovací súbor slúži na vyobrazenie údajov o počtoch prvkov v štruktúrach, taktiež ním overujeme situácie, ktoré môžu vyhodiť chybovú hlášku – pozeráme či vieme nájsť prvok, ktorý sme práve vymazali. Kontrolujeme tieto funkcie na rôznych objemoch datasetov, aby sme si boli istý, že objem dát nevytvorí žiaden problém funkčnosti. Ide o kombináciu implementácií Insert – Search – Delete – Search, ktorý v danej implementácií overuje funkcionalitu Samotných funkcií ako takých, dvojice funkcií, ktoré sú vedľa seba a celkový set kombinácie.

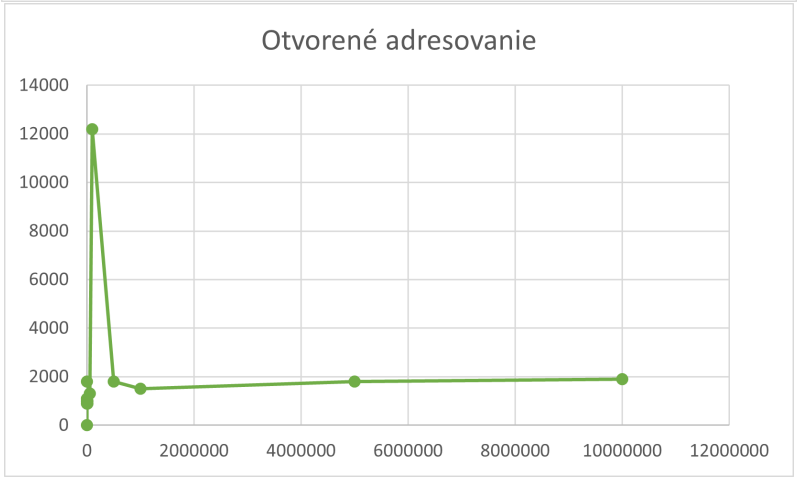
3.2 Časová zložitosť – Insert

Všeobecným teoretickým cieľom bolo dosiahnuť dva výsledky, pre binárne vyhľadávacie stromy priblížiť sa čo najbližšie k logaritmickému priebehu a v rámci hashovacích tabuliek sa priblížiť ku konštantnému priebehu. Z hodnôt ako takých, nevieme jednoznačne určiť daný priebeh kvôli viacerým faktorom, ktoré skresľujú vyhodnocovanie ako napr. veľké odskoky medzi hodnotami, avšak ak tieto hodnoty dáme do grafov, vidíme jasne, že v rámci binárnych stromov sa nám prakticky podarilo dosiahnuť logaritmický priebeh, samozrejme, že existujú vplyvy ako napr. výpočtový výkon, iné zaťaženia hardvéru či softvéru počas merania, ktoré mohli dané hodnoty skresliť. Na záver môžeme konštatovať, že ku chcenému výsledku sme sa priblížili dostatočne blízko a najhoršia situácia, kde by bola efektivita algoritmu rovnaká so spájaným zoznamom je neporovnateľná s našimi výsledkami. Podobné zhodnotenie môžeme konštatovať aj pri hashovacích tabuľkách, kde sme sa snažili dosiahnuť konštantný priebeh, avšak vidíme viaceré nezrovnalosti a body, kde sa rapídne zvyšuje čas – tieto aspekty považujeme za normálne za okolností, že tabuľku bolo potrebné zmenšovať a zväčšovať, čo mohlo zapríčiniť takýto nárast nameraných časov. Na väčšej škále medzi poslednými meranými hodnotami chcene zanedbávame tieto výkyvy hodnôt pri prehashovaní tabuliek na väčšie/menšie a pozorujeme, že v jednotlivých bodoch pri väčších rozdieloch priebeh pôsobí skoro konštante s menšími odchýlkami. Odhliadnuť od priebehov v rámci porovnaní môžeme povedať, že všeobecne v rámci binárnych stromov do počtu 100000 vložených prvkov BST Splay preukazuje značne nižšie časové intervaly na vloženie jedného prvku, avšak v tomto bode nastáva zlom, kde nárast času pri BTS AVL sa pomaly znižuje – môže usúdiť, že pri väčšej kvantite prvkov bolo by vhodnejšie zvoliť AVL, avšak pri menšej početnosti, efektívnejšie v rámci časovej zložitosti by správna voľba bola Splay. Tieto fakty vychádzajú zjavne aj v grafickej podobe z grafu, ktorý porovnáva všetky dátové štruktúry naraz. Pri porovnaní hashovacích tabuliek už nevieme jasne rozdeliť výhodnejšiu efektívnosť na dva stabilné intervaly, teda usudzujeme, že bude rapídne záležať daná efektívnosť na hashovacej funkcií, ktorú sme my zvolili rovnakú pre oba typy tabuliek, taktiež bude záležať od počtu kolízií, teda aj od rôznorodosti vstupných dát. Všeobecne môžeme konštatovať, že najväčší rozdiel v rámci vkladania do hashovacích tabuliek bol v rapídnom náraste pre ich zväčšovaní/zmenšovaní, kde sme pozorovali nárast nameraného času až o 6000 nanosekúnd.

POČET	AVL [ns]	SPLAY [ns]	ADRESOVANIE [ns]	REŤAZENIE [ns]
10	999	3000	1801	1500
50	2699	6500	1099	1100
100	4100	8099	1001	1299
500	5901	9198	900	1101
1000	6901	9898	1100	999
5000	9301	10998	1000	900
10000	11102	18398	900	700
50000	21201	20698	1300	1800
100 000	27701	23098	12200	18900
500 000	30900	25398	1800	1601
1 000 000	34500	28398	1500	1699
5 000 000	38200	31397	1800	1700
10 000 000	41499	34496	1901	2700



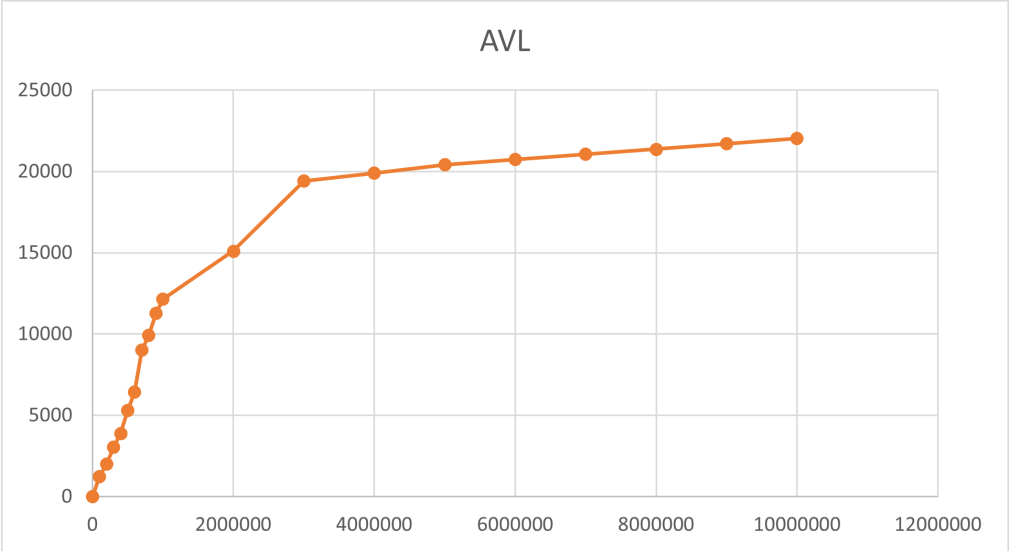


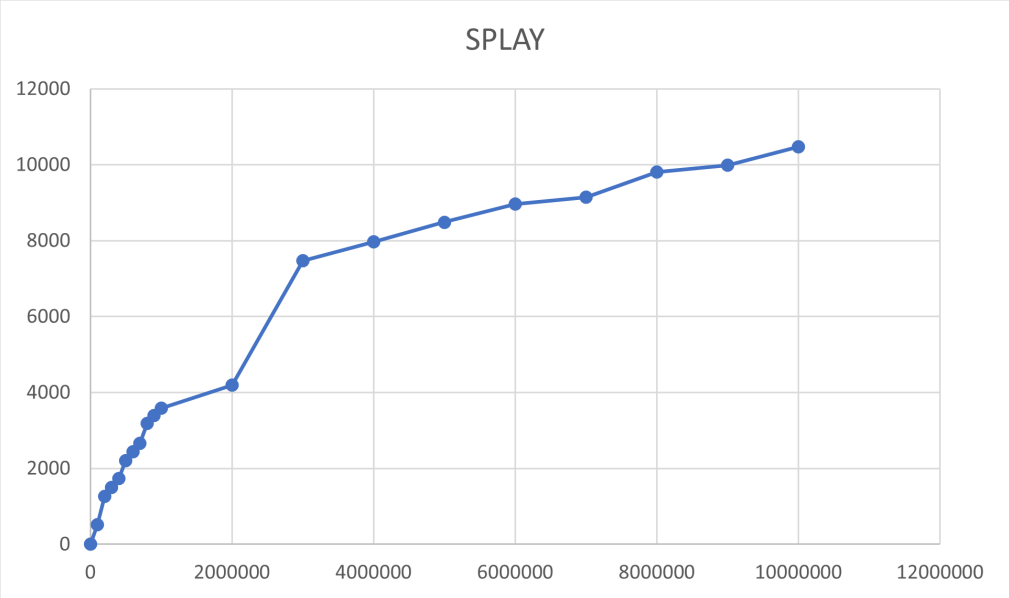


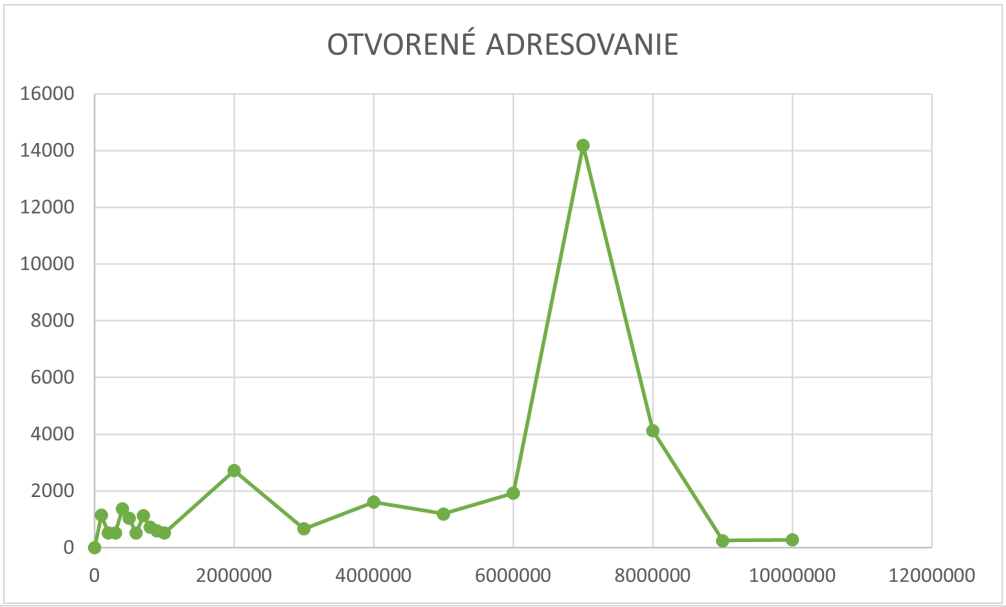
3.3 Časová zložitosť – Search

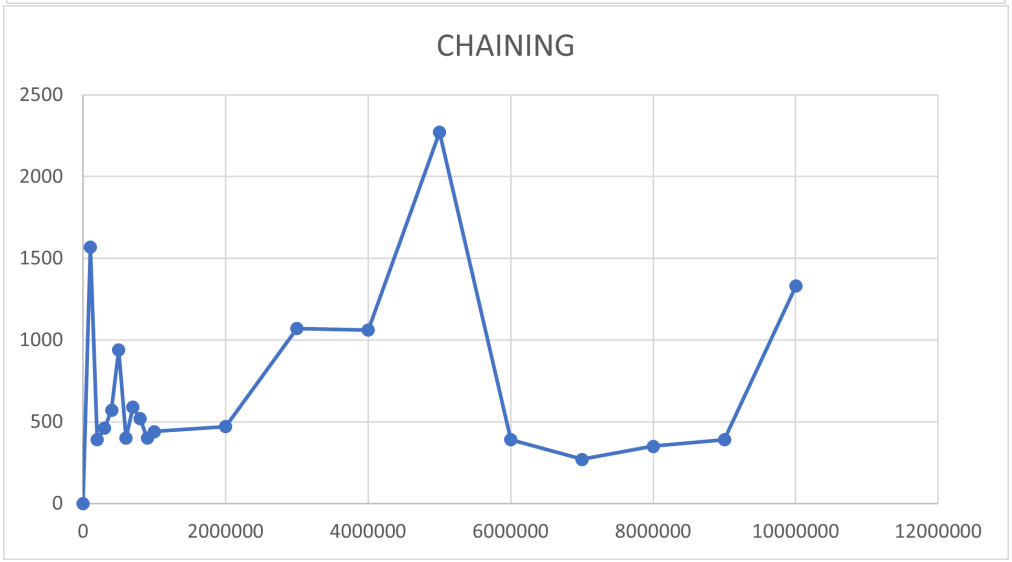
Pri určovaní časovej zložitosti sa veľa zdrojov zhoduje v tom, že časová zložitosť vkladania prvkov by mala mať približne rovnaký priebeh ako časová zložitosť hľadania špecifického prvku, na základe tohto faktu sme sa rozhodli pozorovať priebeh času na menšom kvantitatívnom rozpätí prvkov, ale zato na viacero prestupných bodov, aby sme mohli pozorovať zmeny a výkyvy v priebehu zaznamenávania časových stôp. Taktiež cieľom iného typu merania je potvrdiť logaritmický priebeh binárnych stromov a konštantný priebeh hashovacích tabuliek. Rozhodli sme sa merať od hodnoty 100000 prvkov až po 10000000 prvkov, avšak tak, že každých 100000 prvkov sme zaznamenali čas trvania vloženia ďalšieho prvku do už osadených dátových štruktúr. Zreteľné zmeny sú už z prvého pozorovania najvýraznejšie pri hashovacích tabuľkách, kde na užšom časovom rozmedzí vidíme rôzne výkyvy a malú konštantnosť, avšak vieme, že sa v oboch prípadoch tabuľky zväčšujú a rovnako tak aj riešia kolízie. Usudzujeme, že pri najväčších výchylkách ide o časte generovanie rovnakých indexov v rozmedzí pri vkladaní 300000-500000 prvkov, čo platí pre obe tabuľky. V prípade hľadania daných prvkov v tabuľkách vo všeobecnosti z časových údajov vychádza, že efektívnejšie pracuje metóda reťazenia. V rámci binárnych stromov taktiež vidíme rôzne výkyvy, ktoré môžu byť spôsobené rôznymi aspektami, ktoré boli spomenuté vyššie, avšak poskytnuté dáta jasne naznačujú, že bližšie ku logaritmickému priebehu sa javí byť AVL, ktorý stagnuje na začiatku, ale ďalej sa snaží vytvoriť logaritmické zahnutie. Pri binárnom strome splay si môžeme všimnúť, že hodnoty od určitého bodu začínajú rásť prudšie a odkláňajú sa od logaritmického priebehu. Nadovšetko časové údaje splayu sú pri vyšších hodnotách niekedy aj o polovicu kratšie čo môže znamenať v dlhšom priebehu aj lepšie výsledky, avšak v praktikom použití môžeme konštatovať, že naša implementácia binárneho stromu splay je efektívnejšia v rýchlosti vykonania operácií hľadania prvkov.

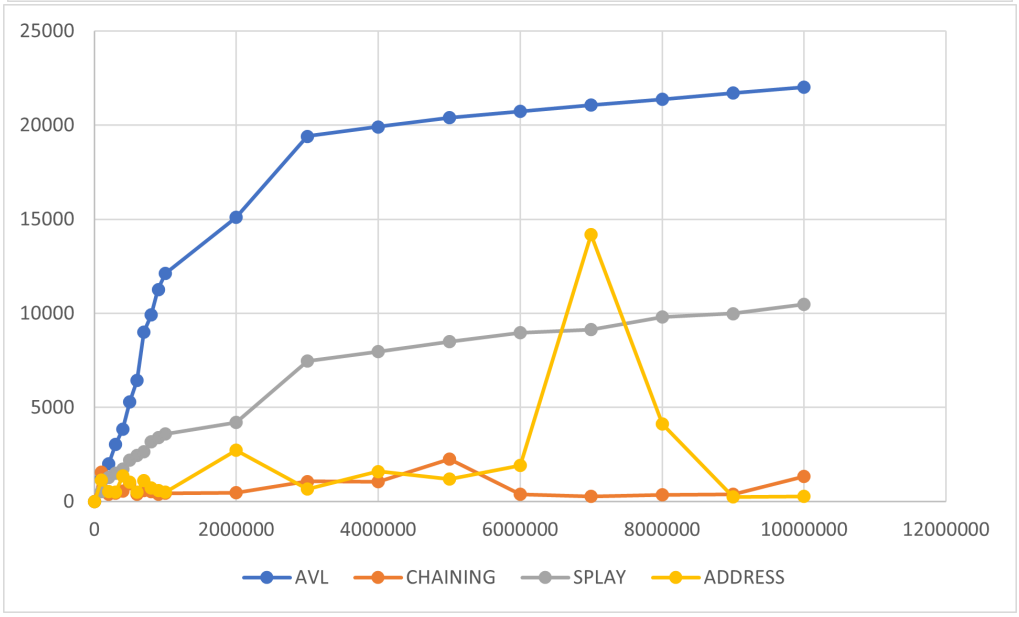
POČET	AVL [ns]	REŤAZENIE [ns]	SPLAY [ns]	ADRESOVANIE [ns]
0	0	0	0	0
100000	1230	1570	520	1150
200000	2010	390	1270	520
300000	3040	460	1500	510
400000	3860	570	1730	1360
500000	5290	940	2210	1030
600000	6430	400	2440	510
700000	9010	590	2660	1130
800000	9910	520	3190	730
900000	11270	400	3400	590
1000000	12130	440	3590	510
2000000	15100	470	4200	2720
3000000	19410	1070	7470	670
4000000	19910	1060	7970	1600
5000000	20410	2270	8490	1190
6000000	20740	390	8970	1920
7000000	21060	270	9150	14190
8000000	21370	350	9810	4120
9000000	21710	390	9990	250
10000000	22020	1330	10480	270






OTVORENÉ ADRESOVANIE
16000
14000
12000
10000
8000
6000
4000
2000
0
0
2000000
4000000
6000000
8000000
10000000
12000000

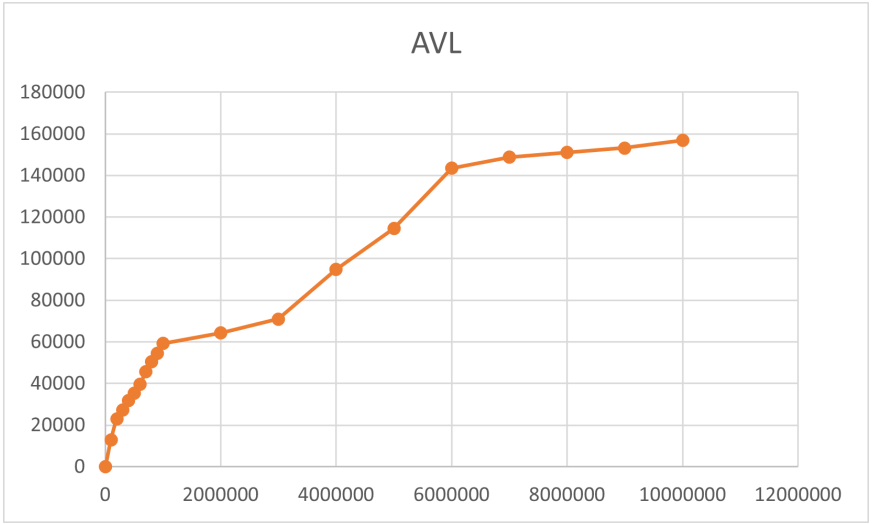

CHAINING
2500
2000
1500
1000
500
0
0
2000000
4000000
6000000
8000000
10000000
12000000

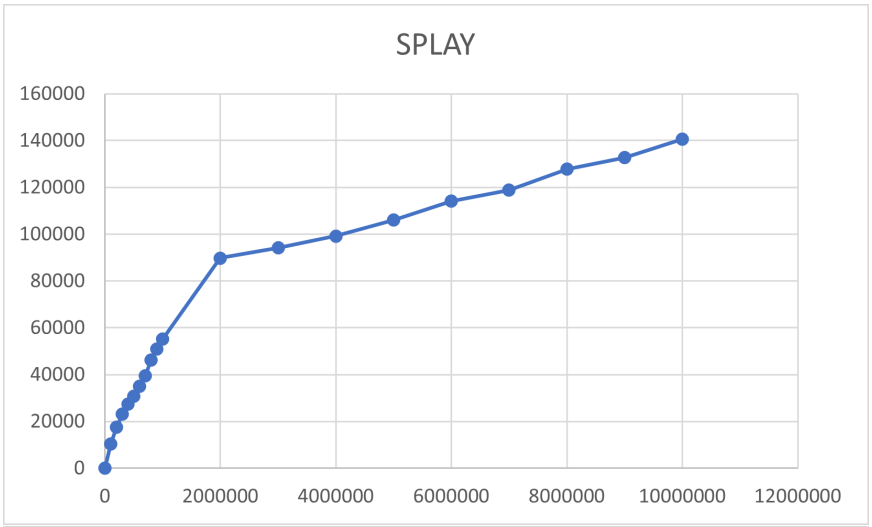

25000
20000
15000
10000
5000
0
0
2000000
4000000
6000000
8000000
10000000
12000000
AVL
CHAINING
SPLAY
ADDRESS

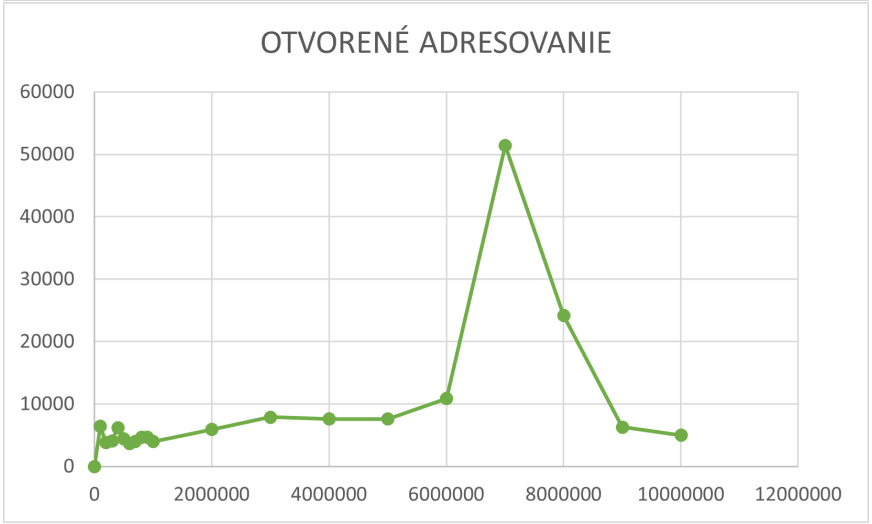
3.4 Časová zložitosť – Delete

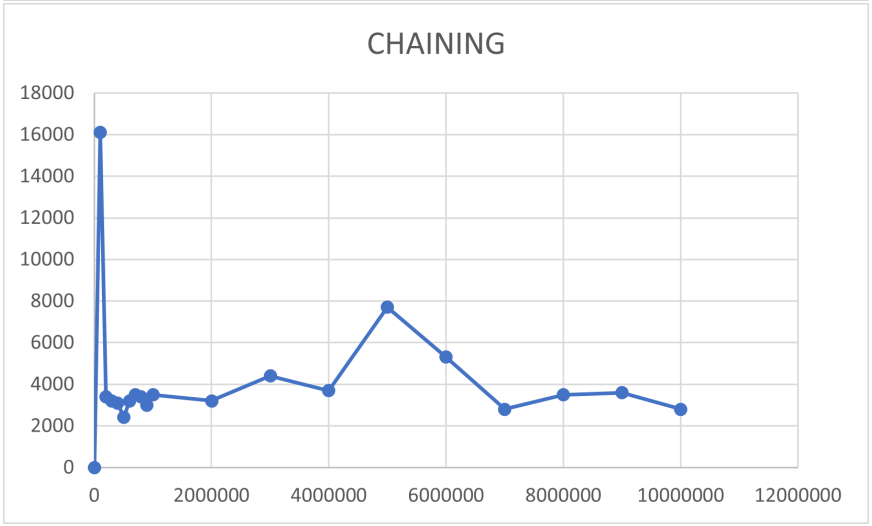
Pri určovaní časovej zložitosti máme v rámci všetkých dátových štruktúr rovnaké očakávania, teda očakávame pre binárne stromy O(log n) a pre hashovacie tabuľky O(1). Postup merania bol totožný s postupom merania pri meraní časovej zložitosti implementácií na hľadanie prvkov. Zaznamenané časové údaje boli rádovo rozdielne aj napriek faktu, že sme použili rovnaký postup zbierania dát. V rámci AVL binárneho stromu sa hodnotovo zvýšili časové záznamy desať násobne, rovnako tak sa zmenili údaje aj v rámci binárneho stromu splay – odhliadnuc od samotných hodnôt, priebehy naznačujú nepresný lineárny priebeh, ako aj v predošlých meraniach. V rámci hashovacích tabuliek hodnotovo dominuje opätovne spôsob reťazenia a obe tabuľky vykazujú relatívne konštantný priebeh. Taktiež po tomto meraní sme schopní nájsť akúsi podobu medzi koncovými časťami grafu pri otvorenom adresovaní (linear probing), kde rapídne rastie čas vykonania operácií pravdepodobne kvôli zmene veľkosti tabuľky. Môžeme zhrnúť, že v rámci celkovej efektivity spomedzi BTS je pri mazaní výhodnejší splay a v rámci hashovacích tabuliek bol o niečo hodnotovo stabilnejší spôsob riešenia kolízií pomocou reťazenia.

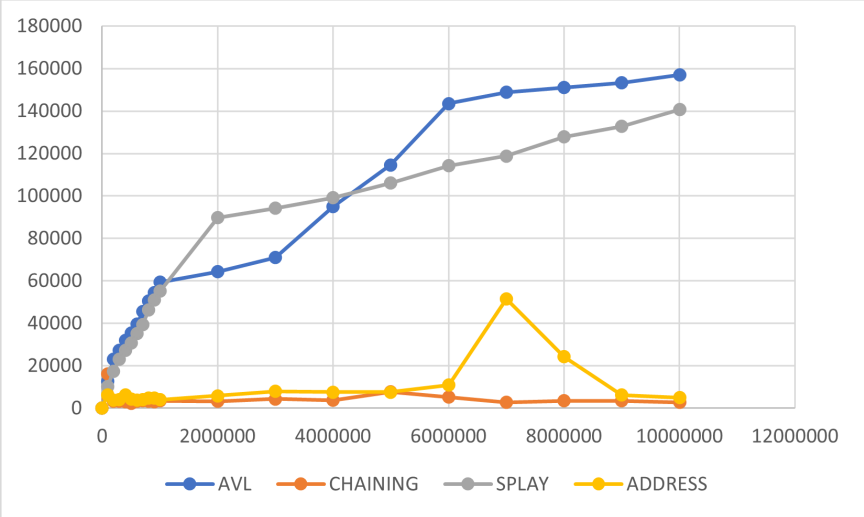
POČET	AVL [ns]	REŤAZENIE [ns]	SPLAY [ns]	ADRESOVANIE [ns]
0	0	0	0	0
100000	12800	16100	10300	6400
200000	23100	3400	17500	3800
300000	27400	3200	23100	4100
400000	31900	3100	27400	6200
500000	35400	2400	30800	4400
600000	39600	3200	35100	3700
700000	45600	3500	39500	4000
800000	50500	3400	46300	4700
900000	54500	3000	50900	4700
1000000	59300	3500	55200	4000
2000000	64400	3200	89800	5900
3000000	71000	4400	94200	7900
4000000	94900	3700	99200	7600
5000000	114600	7700	106100	7600
6000000	143500	5300	114200	10900
7000000	148800	2800	118800	51400
8000000	151100	3500	127800	24200
9000000	153200	3600	132700	6300
10000000	156900	2800	140600	5000




SPLAY
160000
140000
120000
100000
80000
60000
40000
20000
0
0
2000000
4000000
6000000
8000000
10000000
12000000


OTVORENÉ ADRESOVANIE
60000
50000
40000
30000
20000
10000
0
0
2000000
4000000
6000000
8000000
10000000
12000000

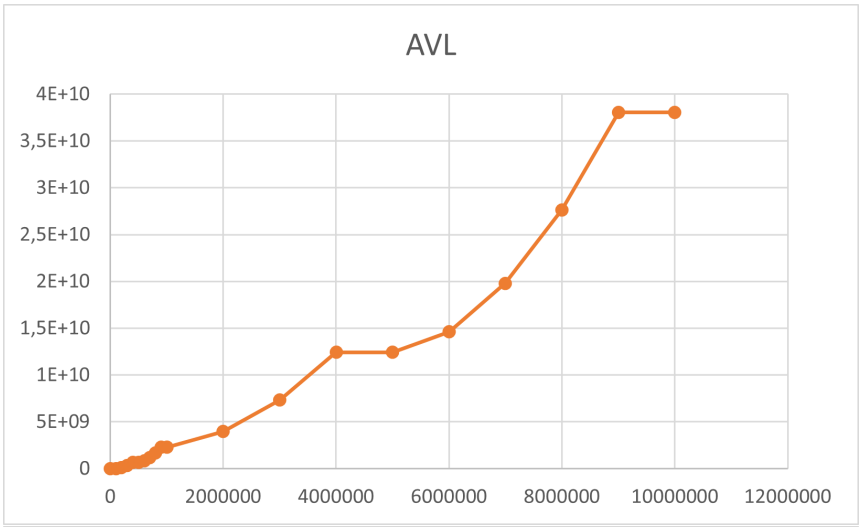

CHAINING
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
2000000
4000000
6000000
8000000
10000000
12000000

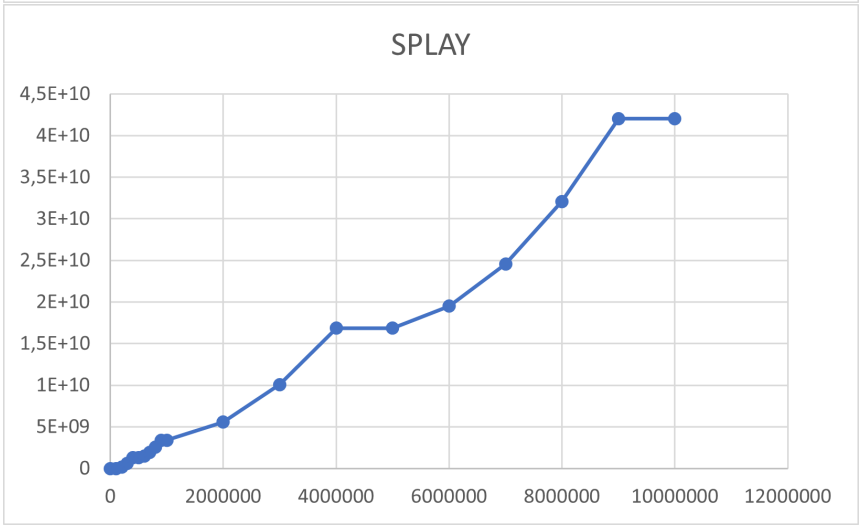


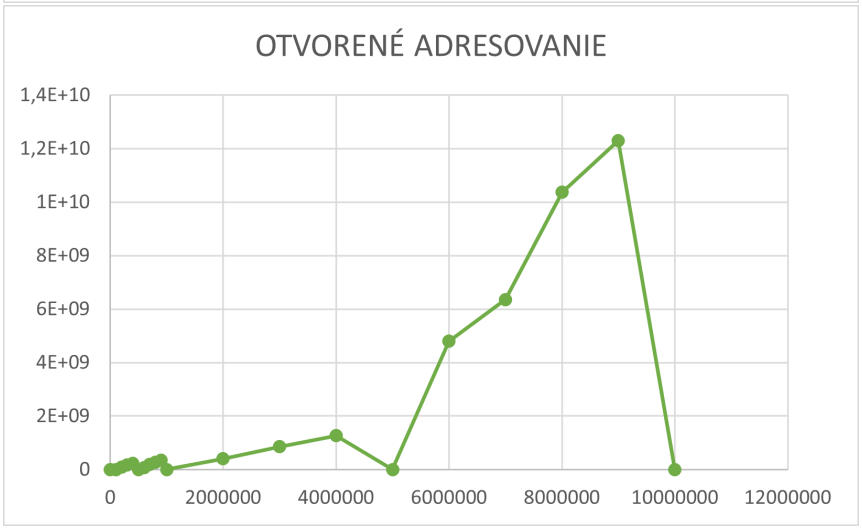
3.5 Časová zložitosť – Insert-Search

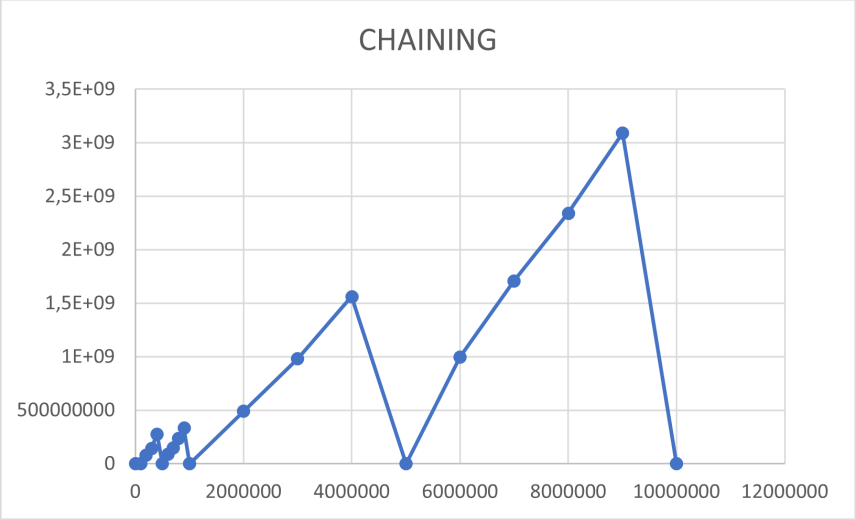
V rámci testovanie tejto kombinácie bolo našim cieľom zistiť ako sa budú správať kombinácie samostatných implementácií – či budú nadobúdať priebeh pôvodných alebo budú nadobúdať úplne iné priebehy. V rámci hodnosti už hovoríme o rádovo väčších číslach z dôvodu, že kombinácie operácií z logického hľadiska by mali trvať dlhšie než samotné implementácie. V rámci samotných priebehov vidíme, že oba binárne stromy zjavne nadobúdajú veľmi podobný priebeh, avšak s rozdielnymi hodnotami s čoho vieme ľahko určiť, že efektívnejší zo stromov bude AVL pre túto danú kombináciu implikácií. Zdanlivo sa vidíme, že priebeh pomaly narastá v rámci časového ohodnotenia a na moment sa stáva konštantným, od tohto bodu sa daný priebeh opakuje. Ak sa pozrieme na hashovacie tabuľky, taktiež majú isté zdieľané črty, kde vidíme isté opakovanie pomalšieho nárastu časového ohodnotenia a od určitého bodu zase naopak rapídne klesá. Taktiež vďaka podobnosti je relatívne ľahké určiť výhodnejšie implementácie z danej dvojice – môžeme konštatovať, že dané výchylky sú podstatne menšie pri metóde reťazenia. Všeobecne môžeme povedať, že naše funkcie stratili pôvodne priebehy.

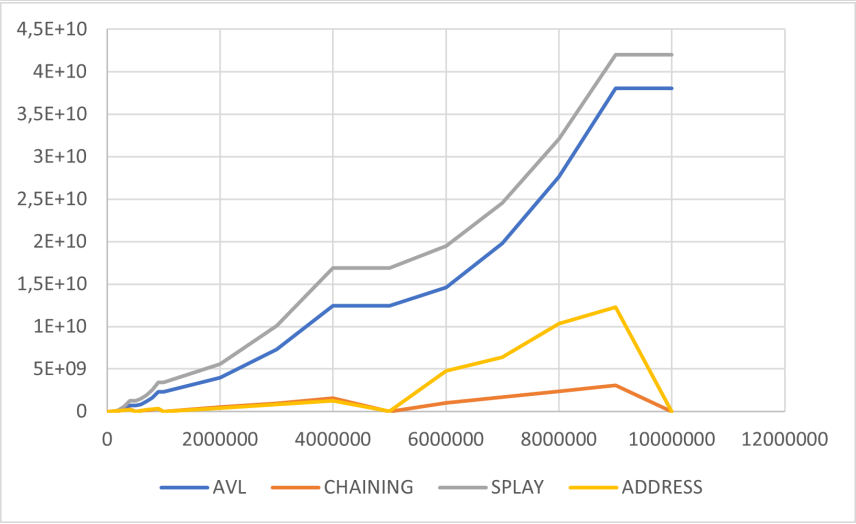
POČET	AVL [ns]	REŤAZENIE [ns]	SPLAY [ns]	ADRESOVANIE [ns]
0	0	0	0	0
100000	10300	20700	8600	25800
200000	89527600	76598500	168120500	85418900
300000	319465000	141508600	580669900	169742000
400000	693342500	276937700	1279141500	233108400
500000	693349300	4400	1279146900	5200
600000	852984900	90232300	1498247900	70496700
700000	1167141100	147114000	1916776400	188391300
800000	1656637500	234844600	2557494600	272644800
900000	2290172200	331977300	3399977100	355647200
1000000	2290181100	4800	3399980700	5600
2000000	3962250300	488544400	5573132600	407580000
3000000	7341235900	978589400	10071292900	851971800
4000000	12429371500	1561591000	16873066600	1264515000
5000000	12429378200	4000	16873081000	11600
6000000	14603465400	997613100	19510007000	4798052600
7000000	19798475200	1708544100	24558064600	6357677700
8000000	27649306600	2341431900	32089727000	10366728600
9000000	38068646700	3089048700	42003744400	12301999300
10000000	38068654300	6600	42003753100	6100


AVL
4E+10
3,5E+10
3E+10
2,5E+10
2E+10
1,5E+10
1E+10
5E+09
0
0
2000000
4000000
6000000
8000000
10000000
12000000


SPLAY
4,5E+10
4E+10
3,5E+10
3E+10
2,5E+10
2E+10
1,5E+10
1E+10
5E+09
0
0
2000000
4000000
6000000
8000000
10000000
12000000


OTVORENÉ ADRESOVANIE
1,4E+10
1,2E+10
1E+10
8E+09
6E+09
4E+09
2E+09
0
0
2000000
4000000
6000000
8000000
10000000
12000000

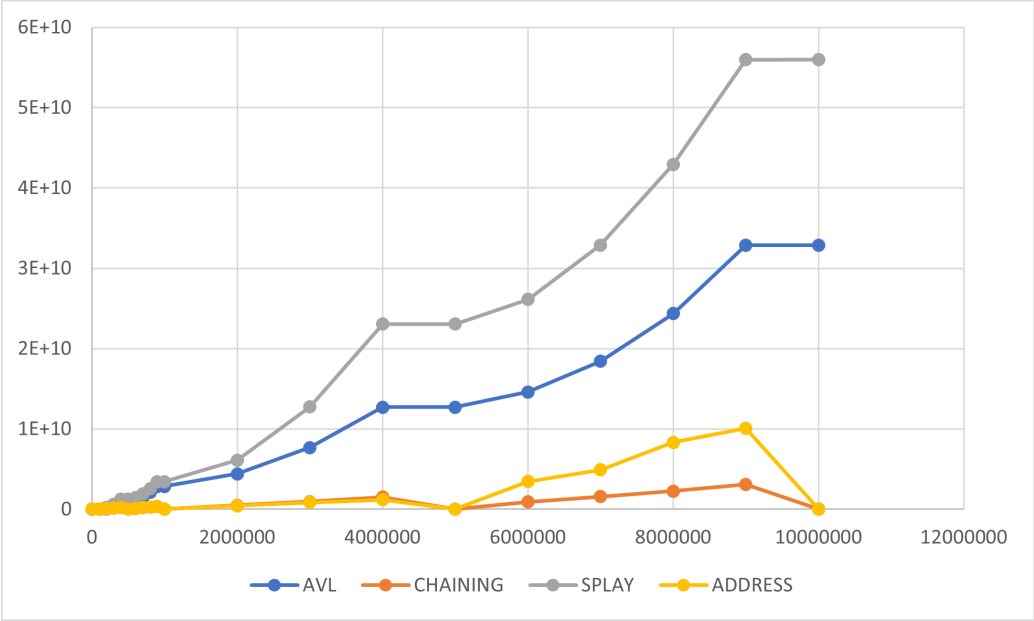




3.6 Časová zložitosť – Insert–Delete

V rámci tejto kombinácie implementácií vieme už na prvý pohľad povedať, že nadobúda veľmi podobné vlastnosti ako kombinácia Insert-Search. Rádovo sa líšia od predchádzajúcej kombinácií implikácií a to isté platí aj medzi hodnosťami jednotlivých funkcií v rámci tejto kombinácie. V jednoduchosti môžeme zhodnotiť, že priebehy sú takmer totožné s predchádzajúcou kombináciou implementácií, taktiež v rámci binárnych stromov je relatívne ľahké určiť efektívnejší variant – opätovne ide o BST AVL, kde samotný rozdiel je ešte hodnotovo viditeľnejší než v predošlej kombinácií. Pri dvojici hashovacích tabuliek taktiež vieme skonštatovať, že efektívnejšia bola metóda reťazenia. Pre jednoduchosť ku danej kombinácií je priložený iba jeden jednotný graf a tabuľka hodnôt.

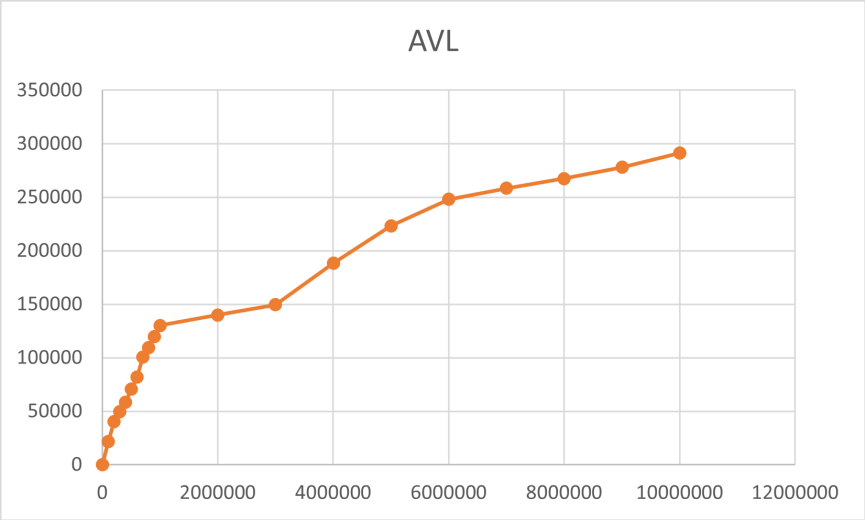
POČET	AVL [ns]	REŤAZENIE [ns]	SPLAY [ns]	ADRESOVANIE [ns]
0	0	0	0	0
100000	8800	21700	16400	14400
200000	147152800	61803800	198983300	67064200
300000	503845900	156983300	625762200	133383600
400000	995941700	261655800	1245462900	219547300
500000	995950700	4700	1245469500	6000
600000	1179773000	113679600	1465064100	87231200
700000	1561238200	200528600	1896293900	204865900
800000	2107169000	265414100	2562216000	296526300
900000	2847941900	381345200	3433064400	381029900
1000000	2847949600	3300	3433072000	4500
2000000	4415488600	509229900	6115883300	438295900
3000000	7695088000	970517200	12745116900	846793700
4000000	12717109200	1513883700	23057468000	1198239800
5000000	12717136000	9500	23057480500	10000
6000000	14607679300	929221100	26147332000	3469270700
7000000	18416021600	1582144900	32895666400	4900081200
8000000	24383004300	2281670200	42954497900	8332060100
9000000	32855496200	3075045400	55977395100	10088459000
10000000	32855513900	5500	55977406600	8500

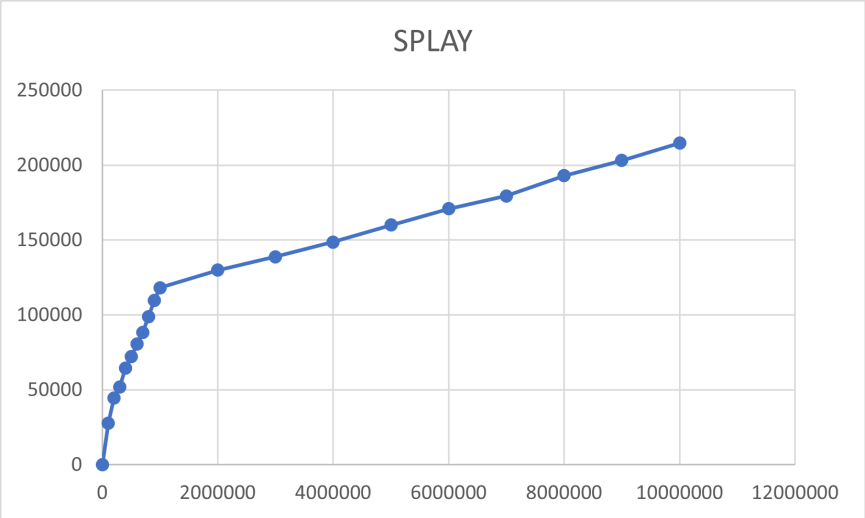


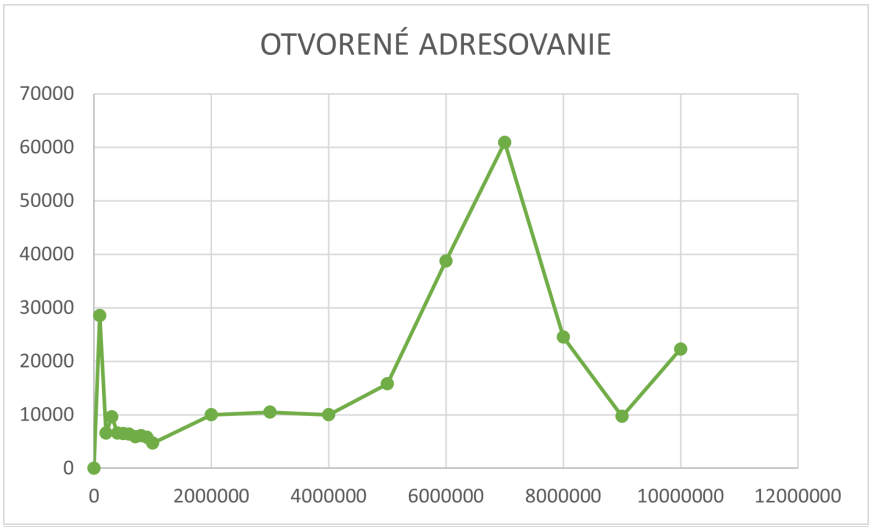
3.7 Časová zložitosť – Insert-Search-Delete

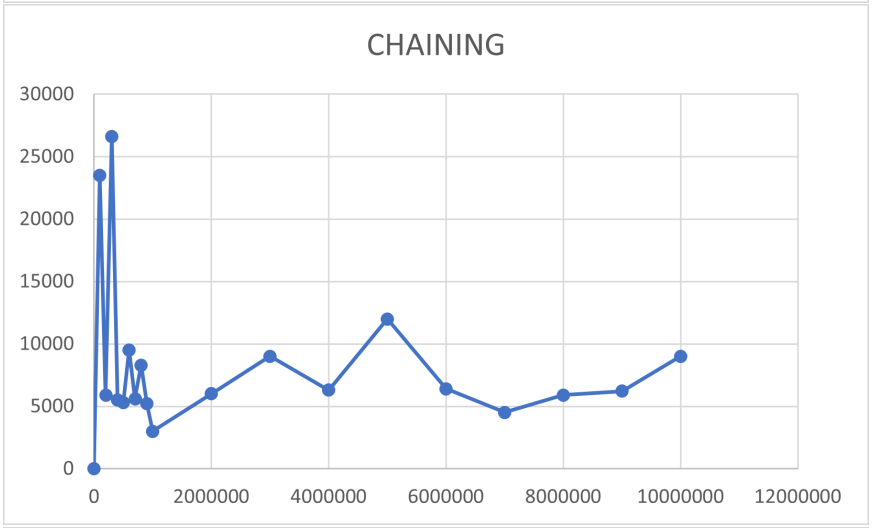
Pri testovaní tejto kombinácie implementácií využívame štandardné počty ako pri predošlých. Môžeme si všimnúť, že tento krát je vždy jeden z dvojice „upravenejší“ a podobá sa veľmi na svoje pôvodné implementácia v zmysle časovej zložitosti. Z dvojice Splay a AVL môžeme jasne určiť, že nakoniec splay vyzerá oveľa podobnejšie pôvodnej O(log n) ako AVL, ktorý podobu stráca pri 300000 vstupoch a jeho čas na operácie sa mení nepravidelne. Avšak je dôležité si povšimnúť, že zo začiatku si sú implementácie veľmi podobné pri nižších hodnotách. Genericky môžeme povedať, že v tomto prípade vhodnejší z týchto dvoch implementácií bude BST Splay. Z dvojice hashovacích tabuliek si môžeme všimnúť, že zo začiatku metóda zreťazenia začala nadobúdať vysoké hodnoty, ktorá sa priebehom ustálili aj napriek zmene veľkosti či kvôli väčšiemu počtu prvkov v jednotlivých reťazcoch. Aj keď sa na prvý pohľad môže zdať, že hodnoty sú oveľa stabilnejšie v prípade otvoreného adresovania (linear probing), hodnotovo sú podstatne vyššie než pri metóde reťazenia, preto optimálnejšie voľba pri tejto kombinácií implementácií by mala byť chaining.

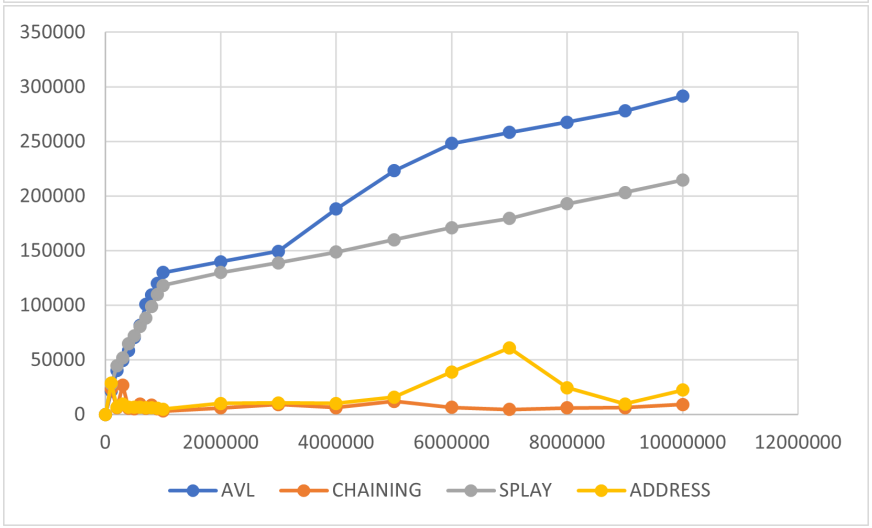
POČET	AVL [ns]	REŤAZENIE [ns]	SPLAY [ns]	ADRESOVANIE [ns]
0	0	0	0	0 100000
21400	23500	27700	28600	
200000	40100	5900	44700	6600
300000	49500	26600	52000	9600
400000	58600	5500	64600	6600
500000	70600	5300	72100	6500
600000	81700	9500	80500	6400
700000	100600	5600	88400	5900
800000	109600	8300	98900	6100
900000	119900	5200	109800	5800
1000000	130000	3000	118100	4700
2000000	139800	6000	129900	10000
3000000	149400	9000	138900	10500
4000000	188100	6300	148700	10000
5000000	223200	12000	160100	15800
6000000	248100	6400	170900	38800
7000000	258300	4500	179500	60900
8000000	267500	5900	192900	24500
9000000	277900	6200	203100	9700
10000000	291300	9000	214800	22300









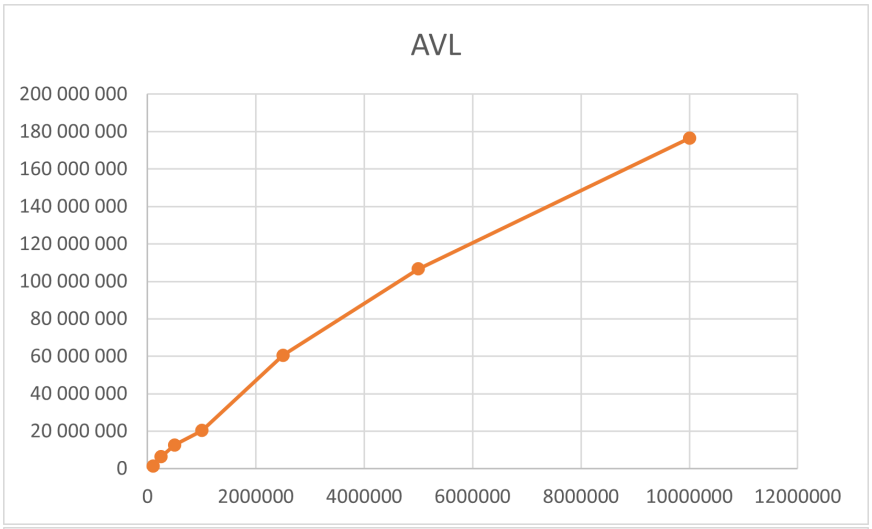


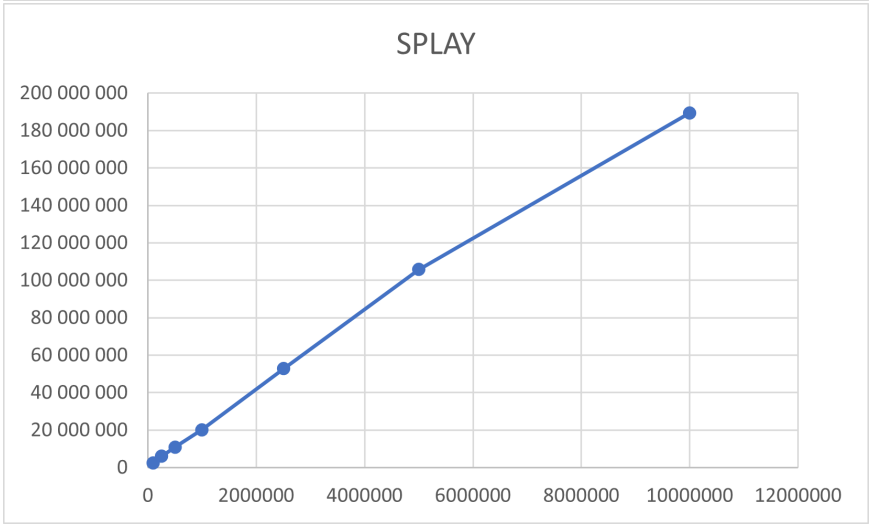
3.8 Priestorová zložitosť – Insert

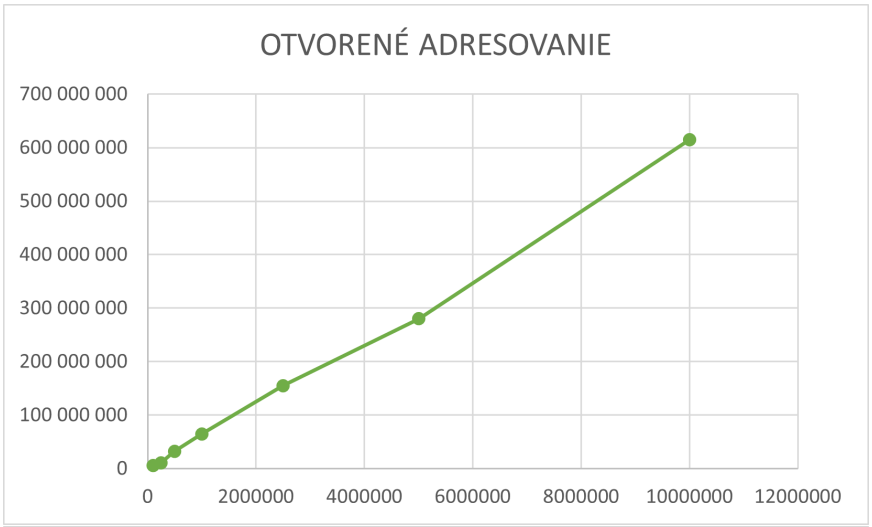
Keď sa jedná o priestorovú zložitosť, chceme zistiť aký objem pamäte si budú dané implementácie zaberať s pribúdajúcimi prvkami v daných štruktúrach. Cieľom je zistiť o aký priebeh sa bude jednať, na získanie dát ohľadom veľkosti pamäte máme nespočetne veľké množstvo softvérov, niektoré sú dokonca implementované v IDE. IDE, ktoré sme použili na tento projekt obsahuje túto funkciu, nato aby sme mali nejaký pevnejšie podklady, je potrebné sledovať taký počet prvkov, ktoré bude daný softvér vedieť zanalyzovať (nič príliš malé). Následne po zvolení počtu vkladaných prvkov sme analyzovali každú jednu zo štruktúr a dáta spracovali v podobe grafu. Pri binárnych stromoch nám vyšli veľmi

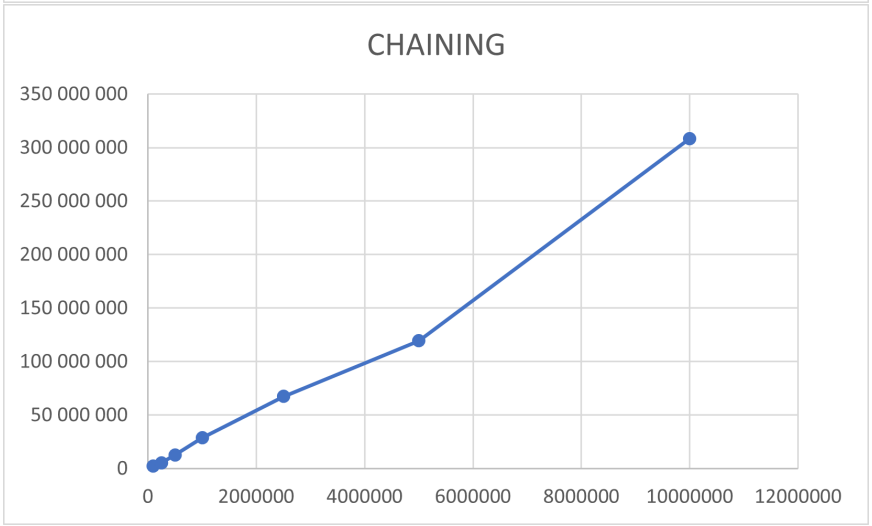
podobné priebehy a to také, že sa odkláňajú od lineárneho priebehu smerom ku funkcie s rýchlejším rastom. Zato naopak hashovacie tabuľky sú tiež relatívne podobné lineárnemu priebehu, avšak ich priebeh sa jemne zakrivuje do pomalšieho nárastu hodnôt. Nakoniec môžeme skonštatovať, že kapacitne najviac zaberajú hashovacie tabuľky a rovnako tak nie sú lepšou voľbou, ak je dôležité kritérium využitá pamäť alebo kapacita všeobecne. Zo všetkých štruktúr v tomto ponímaní má BST AVL.

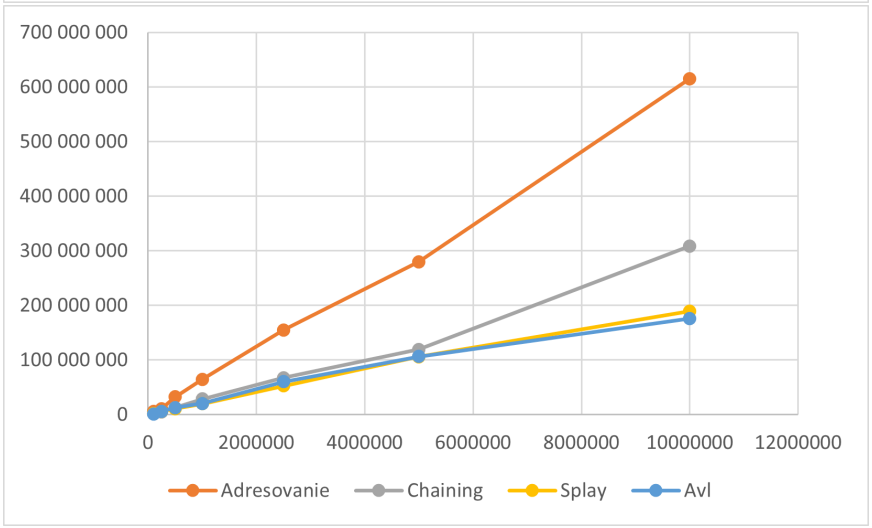
POČET	ADRESOVANIE [B]	REŤAZENIE [B]	SPLAY [B]	AVL [B]
100000	5 807 208	2,408288	2439256	1388640
250000	10748584	5466576	6009520	6326800
500000	32679408	12779848	10867984	12652984
1000000	64660136	28774,32	20094,04	20351712
2500000	154848504	67345184	52866592	60517168
5000000	280122352	119414200	105791088	106656904
10000000	614823472	308429096	189314336	176543344











4 Referencie

Teoretické poznatky som čerpal zo viacerých zdrojov, ktoré boli verejné dostupné. Veľká teoretická časť princípov a operácií daných implementácií mi bola vysvetlená autormi iných kódov či prác na túto tému.
Hashovacie tabuľky: https://iq.opengenus.org/linear-probing/
Vizualizovanie BST: https://www.cs.usfca.edu/ galles/visualization/Algorithms.html
AVL BST: https://www.youtube.com/watch?v=Jj9Mit24CWk
SPLAY BST: https://www.youtube.com/watch?v=qMmqOHr75b8